VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

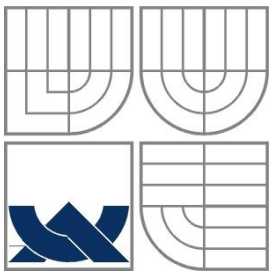
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VÝPOČET VIDITELNOSTI V 3D BLUDIŠTI

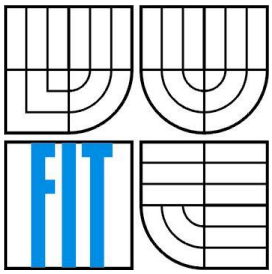
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Vít Hodes
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VÝPOČET VIDITELNOSTI V 3D BLUDIŠTI

VISIBILITY DETERMINATION IN 3D MAZE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Vít Hodes
AUTHOR

VEDOUCÍ PRÁCE Ing. Lukáš Polok
SUPERVISOR

BRNO 2014

Abstrakt

Cílem práce bylo vyhodnocení řešení problému zobrazování velkých uzavřených oblastí v OpenGL pomocí BSP stromů, portálů a výpočtu PVS. Při zobrazování rozlehlého prostředí bludišťového typu to znamená, že vždy většina geometrie nebude viditelná a vykreslovat ji přináší zbytečnou režii. Pokud rozdělíme geometrii světa do hiearchických konvexních jednotek v BSP stromu a vytvoříme mezi nimi portály, můžeme "můžeme spočítat viditelnost a toto vykreslování omezit. S portály mezi listy stromu můžeme spočítat PVS pro každý list a využít toho pro dramatické snížení počtu vykreslovaných listů. Při použití occlusion queries lze toto číslo ještě snížit za cenu pár kreslících příkazů navíc. Jiným přístupem je využití pouze znalostí o portálech, které spojují dva listy, pro postupné testování viditelnosti sousedících listů.

Abstract

Goal of this work was evaluation of ways of solving a problem of displaying large enclosed areas in OpenGL language through means of BSP trees, portals and PVS computation. Having large enclosed maze-like 3D environment means that at any given time most of the geometry won't be visible and drawing them introduces significant unnecessary overhead. Through division of world geometry into hiearchical convex units into BSP tree and creating portals between them it we are able to compute visibility and limit this. With portals between the leaves it is possible to compute PVS for each leaf and use that to dramatically decrease number of leaves drawn. With the usage of occlusion queries this number is futher decreased but at cost of some extra draw commands for queries.Another approach is to use only information about portals which connect two leaves to incrementally test visibility of neighbouring leaves.

Klíčová slova

PVS, BSP, portál, potenciální, sada, viditelnosti, binární, dělení, prostoru, OpenGL, bludiště, 3D, uzel, list, strom

Keywords

PVS, BSP, portal, potential, visibility, set, binary, space, partitioning, OpenGL, maze, 3D, leaf, node, tree

Citace

Hodes Vít: Výpočet viditelnosti v 3D bludišti, bakalářská práce, Brno, FIT VUT v Brně, 2014

Výpočet viditelnosti v 3D bludišti

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením ing. Lukáše Poloka
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Vít Hodes
21.5. 2014

Poděkování

Vedoucímu práce za mnohé připomínky a užitečné rady.

Obsah

1 Teoretický úvod

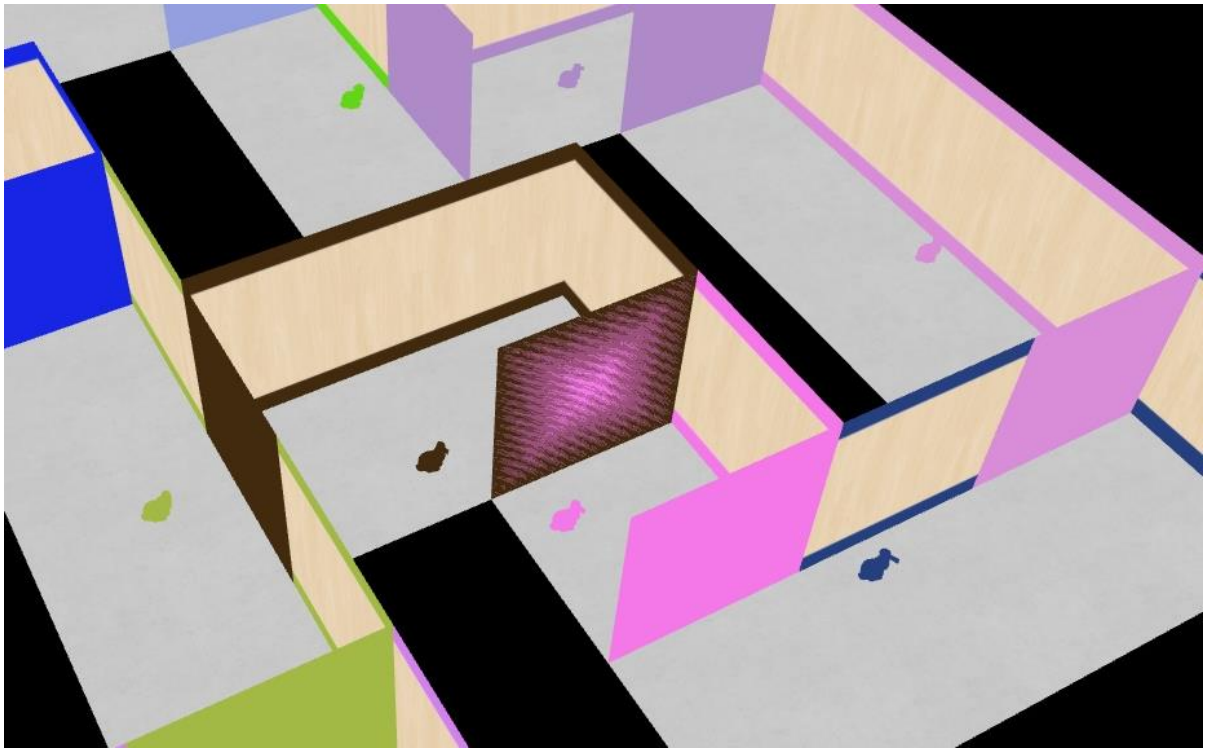
Současné grafické karty zpracovávají data postupně z jednotlivých vrcholů a dalších dat v procesu, jenž lze shrnout pojmem grafická pipeline. Tento proces zjednodušeně probíhá takto. Vrcholy a jiná data jsou interpretovány jako proud trojúhelníků, které se transformují do prostoru obrazovky. Trojúhelníky, které spadají mimo hranice obrazového prostoru, se ořízno (viewport culling) a zbylá geometrie postupuje dál do rasterizéru, kde se převede na jednotlivé vzorky. Ty se filtrují podle své hloubky tak, že jen nejbližší zůstává (tzv. depth test). Následně se provedou operace nad každým vzorkem ve fragment shaderu, jako výpočty efektů, světla apod. Na konci se zpracované fragmenty vloží do framebufferu, ze kterého se výsledný obraz dostává na monitor. Náročnost operací v grafické pipeline může být jen jedním faktorem omezujícím běh aplikace.

Dalším limitujícím faktorem může být počet draw calls (volání kreslících funkcí). V dnešní době jsou typicky všechna data použitá při kreslení scény v paměti přímo na GPU a tak počet draw calls přicházejících z procesoru do určité míry zatěžuje sběrnici. Pokud je těchto volání příliš, aplikace přestává běžet efektivně a výkon se propadá.

I přesto, že moderní grafické karty mají každý rok větší výkon, vykreslovat vždy vše co scéna obsahuje je u náročných scén naprosto nemožné, jak z pohledu počtu draw calls tak z pohledu množství geometrie a jiných zdrojů aplikace, které se musí vejít do paměti GPU. Grafická karta ale nemá vůbec žádný vliv na to, jaké množství dat se do ní posílá a tak je na programátorovi aplikace, aby se postaral o to, aby se po grafické kartě nechtělo kreslit celou komplexní scénu a práci jí trochu zjednodušil. Toho se dosahuje různými způsoby, z nichž některé jsou dále implementovány a prozkoumány.

1.1 Problém viditelnosti a Z-buffer

Dalším problémem kromě množství geometrie je problém viditelnosti jednotlivých vzájemně se překrývajících objektů. Toto pomáhá řešit depth buffer (nebo také Z-buffer), což je v podstatě pole kopírující rozsahem množství zobrazovaných pixelů ve kterém se uchovává informace o vzdálenosti vzorku/pixelu. Pokud na jednu pozici v tomto bufferu připadá více vzorků, porovná se z hodnota nově příchozího se zapsanou hodnotou a zapíše se jen v případě, že jeho z hodnota je nižší než dříve zapsaná. Toto se nazývá depth test. Tato fáze umožňuje rychle eliminovat pixely, které stejně nebudou viditelné a ušetří se nákladné výpočty texturování, osvětlování a provoz pixel shaderů.



Obrázek 1 - Z-fighting

Z-buffer si přes své nesporné výhody s sebou nese i určité technické omezení. V závislosti na bitové šířce může dojít u dvou blízkých objektů či ploch k takzvanému Z-fighting, kdy část jednoho objektu je klasifikována jako bližší zatímco místo druhé části prvního objektu se zobrazí druhý objekt. Proto se v současnosti nepoužívají šířky menší než 16 bitů a nejvyšší v současnosti podporovanou hodnotou je šířka 32 bitů. Nejčastěji se ale využívá šířky 24 bitů s 8 bity pro stencil buffer nebo nevyužitých jako padding. Dalším faktorem omezujícím tento jev je skutečnost, že vzorkování mezi vzdálenostmi blízké a vzdálené clipping plane (roviny ohraničující prostor ve kterém se nacházejí viditelné objekty) není rovnoměrné. Rozlišení nízkých hodnot je mnohem jemnější než hodnot vzdálených, což většině aplikací velice vyhovuje (přes 90% přesnosti pokrývá prvních 10% vzdálenosti). Navíc vzdálené objekty jsou velmi pravděpodobně již minimálně částečně překryté objekty blízkými a tvoří jen malou část výsledného obrazu.

Problém viditelnosti šel před univerzální dostupností Z-bufferu řešit například seřazením zobrazované geometrie a vykreslováním od nejvzdálenějších po nejbližší – tzv. Painter's algorithm (malířův algoritmus). K jeho nevýhodám ale patří časté překreslování pixelů a vykreslování objektů, jež nelze jednoznačně seřadit podle jejich vzdálenosti od kamery.

Protože standard určuje, že u výsledku se musí jevit že depth test proběhl až po zpracování ve fragment shaderech, nechává toto prostor pro určité optimalizace, které umožňují v určitých případech úsporu pomocí tzv. early fragment test. Toto probíhá zcela v režii GPU a nelze vynutit přes OpenGL. Hlavním kritériem pro tuto optimalizaci je, že se nesmí zapisovat do proměnné gl_FragDepth ve fragment shaderech.

Early fragment test (1) lze na nejnovějším hardwaru vynutit pomocí layout modifikátoru ve fragment shaderech, ale stále platí, že výsledek se nesmí lišit od případu, kdy by se depth test provedl až po projití vzorku fragment shaderem. To znamená, že veškeré modifikace proměnné gl_FragDepth budou ignorovány.

Existuje ještě conservative depth test (2), který se používá podobným způsobem jako early fragment test a říká grafické knihovně jakým způsobem budou příchozí fragmenty modifikovány. Je možné zvolit, že hloubka bude jen větší, menší (např. bump mapping), beze změny nebo bez garancí, což je defaultní hodnota. Toto umožní grafické knihovně určité optimalizace a přeskládání příkazů, což může zvýšit výkon.

Hiearchical depth test (3) je technika pro ještě rychlejší odmítání neviditelných pixelů. Spočívá v tom, že se po vykreslení všech velkých objektů blokujících viditelnost do menšího depth bufferu (např jen 256x128). Z této textury se vytvoří mipmapy a při kreslení objektů ve scéně se v shaderu podle velikosti kresleného objektu vybere vhodná úroveň (velký objekt vybírá z menších mipmap částí, malý objekt z větších).

1.2 Popis BSP stromu

BSP neboli binary space partitioning (binární dělení prostoru) je metoda pro hiearchické rozčlenění prostoru do základních konvexních jednotek, které tvoří listy stromu. Uzel stromu tvoří vždy rovina podle které jsou potomci rozděleni, takže nalezení odpovídajícího uzlu ke konkrétní pozici obnáší postupné porovnání, je-li vpravo či vlevo od dělící roviny uzlu. Tuto strukturu lze využít ke kreslení, kdy se rekurzivně prochází stromem v pořadí daném výsledkem porovnání, ale samotnou ji nelze využít k odstranění kreslení momentálně neviditelných listů. Samotný BSP strom lze použít pro kreslení bez využití Z-bufferu, kdy se využívá skutečnosti, že jednotlivé uzly a listy stromu jsou seřazeny podle své polohy. Lze tak projít stromem od nejvzdálenějších uzlů k nejbližším a tak vyřešit problém viditelnosti tímto způsobem. Tento způsob ale neřeší přílišné zatěžování grafické karty neviditelnou geometrií.

1.2.1 Datová struktura BSP strom

Následuje příklad datové struktury BSP strom, která tvoří základ další implementace. Jde o klasický binární strom s kořenovým uzlem ve struktuře stromu a levým a pravým potomkem v každém uzlu. Nejdůležitější odlišností od prostého stromu je polygon dělicí roviny, podle kterého se pak lze orientovat při procházení stromu.

```
1  struct BSPTree {

2     BSPTreeNode RootNode
3  }


4  struct BSPTreeNode {

5     BSPTree Tree

6     BSPTreePolygon Divider
7     BSPTreeNode LeftChild
8     BSPTreeNode RightChild

9     list PolygonSet
10 }
```

Kód segment 1 – struktura BSP strom

1.2.2 Generování BSP stromu

Původní návrh konceptu dělení prostoru prezentoval Henry Fuchs se spoluautory k urychlení řazení geometrie pro řešení problému viditelnosti (4).

Tato rekurzivní funkce se stará o vygenerování stromové struktury BSP stromu. Na začátku se zavolá s argumentem, kterým je kořenový uzel obsahující v `polygonSet` veškerou geometrii vytvořeného bludiště či jiného prostředí.

Na začátku se ověří, zda není náhodou tento uzel už konvexní a nemělo by smysl se jej pokoušet dál dělit. Pokud je konvexní, uzavírá se tato větev rekurze. Dál následuje nejdůležitější část tohoto procesu a to nalezení nejvhodnějšího trojúhelníku/roviny, která dělí stávající množinu trojúhelníků, pokud možno, na poloviny. Tato funkce je v detailu rozebrána dále.

Po nalezení hledané dělící roviny se tato uloží do uzlu a přikročí se ke kategorizaci geometrie uzlu. Pozice každého trojúhelníku se porovná s dělicí rovinou a vloží se do jedné ze dvou množin - `positiveSet, negativeSet.` Pokud tuto množinu trojúhelník protíná, je nutné jej rozdělit a vložit každou část zvlášť. Trojúhelníky na dělicí rovině se vloží do pozitivní množiny.

Nakonec se z těchto množin vytvoří nové uzly tvořící oba potomky stávajícího uzlu, které tvoří nové argumenty této funkce. Užívá se konvence, že za kladný směr se považuje pozice vpřed a odpovídá pravému potomku stromu. Takže `positiveSet` je základem pravého potomka a `negativeSet` levého potomka.

```
1  generateBSPTree(Node)

2     if isConvex(Node)
3        return

4     divider = chooseBestDividingPolygon()

5     list positiveSet, negativeSet
6     polygon front, back

7     foreach polygon in polygonSet

8        side = calculateSide(divider, polygon)

9        if side is INFRONT
10          positiveSet.add(polygon)

11       if side is BEHIND
12          negativeSet.add(polygon)

13       if side is SPANNING
14          splitPolygon(polygon, front, back)
15          positiveSet.add(front)
16          negativeSet.add(back)

17       if side is COINCIDING
18          positiveSet.add(polygon)

19    Node.RightChild = new Node(positiveSet)
20    Node.LeftChild = new Node(negativeSet)

21    generateBSPTree(Node.RightChild)
22    generateBSPTree(Node.LeftChild)
```

Kód segment 2 – generování BSP stromu

1.2.3 Hledání nejlepší dělicí roviny

Tento algoritmus je výpočetně nejnáročnější částí generování BSP stromu. Zvláště při velkém množství geometrie bludiště začíná být čas strávený v této funkci velmi znatelný(viz graf v sekci 3.2).

```
1 chooseBestDividingPolygon(polygonSet)
2    isConvex(polygonSet)
3       return NOPOLYGON

4    minRelation = 0.8f;
5    bestPolygon = NOPOLYGON;
6    leastSplits = INT_MAX;
7    bestRelation = 0.0f;
8    minRelationScale = 2.0f;
9    relation = 0.0f;

10   while bestPolygon is NOPOLYGON

11      foreach polygon1 in polygonSet

12         if polygon1 used as divider
13            continue

14          positive = 0, negative = 0, spanning = 0

15          foreach polygon2 in polygonSet
16             side = calculateSide(polygon1, polygon2)

17             if side is INFRONT or COINCIDING
18                positive++
19             if side is BEHIND
20                behind++
21             if side is SPANNING
22                spanning++

23           if(positive < negative)
24              relation = positive / negative
25           else
26              relation = negative / positive

27           if(relation >= minRelation and
                (spanning < leastSplits or
                 (spanning == leastSplits and relation > bestRelation)))
28              bestPolygon = i
29              leastSplits = spanning
30              bestRelation = relation

31      minRelation = minRelation / minRelationScale;

32 return bestPolygon
```

Kód segment 3 – hledání nejlepší dělicí roviny

Na začátku algoritmu (5) se ověří, zda nejdeme hledat dělicí rovinu v konvexní množině, což by bylo zcela zbytečné a pak následuje několik definic. Důležité jsou `minRelation` a `minRelationScale`. První má vliv jaký poměr mezi počty trojúhelníků na kladné a záporné straně budeme považovat za přijatelný a druhá určuje, o kolik ze svých nároků slevíme v případě, že po prvním průchodu nenalezneme uspokojující výsledek.

Samotný algoritmus probíhá, dokud není nalezen vyhovující trojúhelník/rovina. První na začátku po řadě vybíráme kandidátní polygony z množiny polygonů uzlu. Vynulujeme si sledované parametry, začneme porovnávat s veškerou geometrií uzlu a počítat kolik trojúhelníků se nachází před, za a kolik jich rovinu kříží. Ze zjištěných parametrů se spočítá jejich vzájemná relace. Čím jsou počty více stejné, tím se spočítaná relace blíží číslu 1.0f.

Na závěr, pokud je spočítaná relace vyšší než původních 0.8f a zároveň by si použití této dělící roviny vynutilo méně rozdělování trojúhelníků, nebo pokud je počet rozdělení alespoň stejný a tato rovina má ještě lepší relaci než případná dříve nalezená, našli jsme novou nejlepší dělicí rovinu. Takto vyzkoušíme jako kandidátní dělicí rovinu všechny trojúhelníky v uzlu a pokud jsme našli alespoň jeden, funkce se vrací s vhodným trojúhelníkem pro dělicí rovinu.

1.3 Portály

Po vygenerování BSP stromu se vytvoří portály. Každý list obsahuje konvexní část geometrie světa, která navazuje na další a společně tvoří veškerou množinu vší geometrie. Tyto části byly vytvořeny pomocí dělících rovin, teď uložených v uzlech, a tyto můžeme využít pro vygenerování polygonů, které tyto konvexní listy oddělují. Tyto polygony se nazývají portály.

Při průchodu stromem se v každém uzlu, kde dělící rovina nebyla použita ke generování portálu, vytvoří polygon, který lehce přesahuje bounding box (set extrémů souřadnic) geometrie tohoto uzlu a s orientací dělící roviny. Tento polygon se pošle oběma potomkům. Dále se v každém dalším uzlu stromu polygon rozřízne dělící rovinou a pošle se hlouběji, každá část jednomu potomku. Po dosažení listu stromu se polygon ořeže ještě geometrií uzlu, odstraní se překrývající části a zbylý polygon se uloží.

Po dokončení tohoto procesu se v každém uzlu nachází v místě spojení se sousedním uzlem polygon, který má svůj protějšek na stejné pozici, ale v sousedním uzlu. Využitím této informace získáme první základní poznatky o vzájemném prostorovém vztahu uzlů.

Na bázi portálů existují vykreslovací způsoby využívající informace o sousednosti listů a ořezávání view frustum (pohledového hranolu) na velikost viditelných portálů. My je ale využijeme jinak a to pro předpočítání PVS.

1.3.1 Proces umisťování portálů

Zde se v uzlech generují portály a umísťují se do listů stromu (5). Pokud je aktuální uzel listem, příchozí portál se ořízne polygony v listu. Pokud k ořezání nedošlo, portál už byl ořezán veškerou geometrií a můžeme si ho uložit. Pokud listem není a dělicí rovina ještě pro vygenerování portálu nebyla použita, vytvoří se portálový polygon lehce přesahující bounding box aktuálního uzlu a pošle se pravému a levému podstromu. Dál se pokračuje snahou poslat portálový polygon stromem níže. V případě, že tento polygon kříží dělicí rovinu aktuálního uzlu, rozdělí se a části se pošlou shora stromu.

```
1 placePortals(portalPolygon, Node)

2    if not isLeaf(Node)
3       if Node.divider is not used as portalPolygon
4          portal = Node.getPortal()
5          placePortals(portal, Node.LeftChild)
6          placePortals(portal, Node.RightChild)

7       side = calculateSide(portal, Node.divider)

8       if side is INFRONT
9          placePortals(portalPolygon, Node.RightChild)
10      if side is BEHIND
11         placePortals(portalPolygon, Node.LeftChild)
12      if side is COINCIDING
13         placePortals(portalPolygon, Node.RightChild)
14         placePortals(portalPolygon, Node.LeftChild)
15      if side is SPANNING
16         polygon front, back
17         splitPolygon(portalPolygon, Node.divider, front, back)
18         placePortals(front, RootNode)
19         placePortals(back, RootNode)

20   else
21      isClipped = false

22      for each polygon in Node
23         side = calculateSide(portal, polygon)

24         if side is SPANNING
25            isClipped = true
26            splitPolygon(portalPolygon, polygon, front, back)
27            placePortals(front, RootNode)
28            placePortals(back, RootNode)

29      if not isClipped
30          Node.addPortal(portalPolygon)
```

Kód segment 4 – umisťování portálů

Když uzel není listem a portál neprotíná dělicí rovinu, pošle se níže stromem v závislosti na své poloze vůči dělicí rovině uzlu. Pokud je vepředu, je poslán pravému potomku, pokud vzadu, levému. V případě že portálový polygon tvoří stejnou rovinu s dělicí rovinou, musí být poslán oběma potomkům.

Pokud dojde portálový polygon do listu, ořeže se postupně geometrií listu a pošle znova seshora stromu. Když k ořezání nedojde, je uložen v listu. Konečným stavem je, že ze všech dělicích rovin uzlů byly vygenerovány portálové polygony, které se dělily všemi dělicími rovinami po cestě a skončily v listech.

1.4 PVS

PVS neboli potential visibility set (množina potenciální viditelnosti) je množinou všech listů, které je možné vidět z jednoho konkrétního listu. To znamená, že pokud máme tuto množinu zjištěnou pro každý list BSP stromu a tuto množinu využijeme ke kreslení, eliminovali jsme zbytečné vykreslování listů, které nikdy nemohly být vidět z daného uzlu.

Tuto množinu získáme tak, že v první fázi porovnáme všechny portály se sebou navzájem a ty co sdílejí prostor, si poznačí list svého protějšku. Tyto listy jsou bezprostředními sousedy daného listu a tvoří základ PVS. Další část je náročnější, co se týče výpočetní složitosti.

Když známe bezprostřední sousedy každého listu, víme, že tito sousedé jsou z daného uzlu vždy potenciálně viditelní. Teď se tuto množinu musíme pokusit rozšířit o listy z množin svých sousedů. To uděláme tak, že pokud jsou viditelné jejich portály, jsou navzájem viditelné i listy v nichž se nacházejí. Toto se ověřuje pomocí paprsků (nebo spíše úseček) mezi těmito portály. Tento paprsek nesmí protnout žádnou geometrii listů mezi těmito dvěma listy včetně listů, které se můžou nacházet mezi nimi. Je tedy nutné testovat všechny listy v momentálním PVS.

Často ale jeden paprsek nemůže stačit a je proto nutné provést tuto kontrolu pro více bodů rozprostřených po portálech, a aby bylo možné s určitou mírou jistoty prohlásit, že listy nejsou navzájem viditelné, musí se vyzkoušet všechny kombinace paprsků oproti vší geometrii momentálního PVS. Tato operace může být v závislosti na velikosti listů a PVS poměrně náročná, ale použití PVS zaručí, že se nebude při kreslení ztrácet čas vykreslováním listů, které nemohou být viditelné. Výpočtem PVS pomocí protínání portálů se věnoval Seth Teller (6).

1.4.1 Výpočet viditelnosti

V první části tohoto algoritmu (7) (8) se najdou páry portálů, které sdílejí stejný prostor, tj. jsou spojené přes portály. Když známe bezprostřední sousedy všech listů, můžeme pokročit dál a pokusit se množinu `PVS` co nejvíce rozšířit. Takže se po řadě pro všechny listy prochází jejich PVS. Pro ještě nekontrolované sousedy těchto listů se provede test na viditelnost. Pokud je list viditelný ze současného listu, vloží se vzájemně do množiny potenciálně viditelných listů. Na konci máme kompletní seznam potenciálně viditelných listů v každém listu stromu.

```
1 traceVisibility(Tree)

2    for each leaf1 in Tree
3       for each leaf2 in Tree
4          if isConnected(leaf1, leaf2)
5             leaf.addToPVS(leaf2)


6    for each leaf1 in Tree
7       for each uncheckedLeaf in leaf1.PVS
8          for each leaf2 in uncheckedLeaf.neighbours
9             if not leaf1.PVS.contains(leaf2) and isVisible(leaf1, leaf2)
10               leaf1.PVS.add(leaf2)
11               leaf2.PVS.add(leaf1)
```

Kód segment 5 – výpočet viditelnosti

1.5 Způsoby využití BSP a PVS ke kreslení

Byly implementovány čtyři základní vykreslovací způsoby + dva odvozené speciálně pro měření hodnoty overdraw (poměrný počet vzorků prošlých depth testem oproti rozlišení okna).

1.5.1 BSP

Tento způsob spočívá v procházení BSP stromu od nejbližšího listu po nejvzdálenější. Společně s depth bufferem jde o poměrně efektivní způsob, co se týče overdraw, a pro méně rozlehlé a detailní geometrie jde o funkční řešení.

Obrázek 2 - kreslení pomocí BSP stromu

Na ilustračním obrázku je vidět bludiště vykreslené celé. Jde o bludiště s 36 listy a v každém listu se kreslí i králík, takže celkem 72 volání funkce typu `glDraw*`.

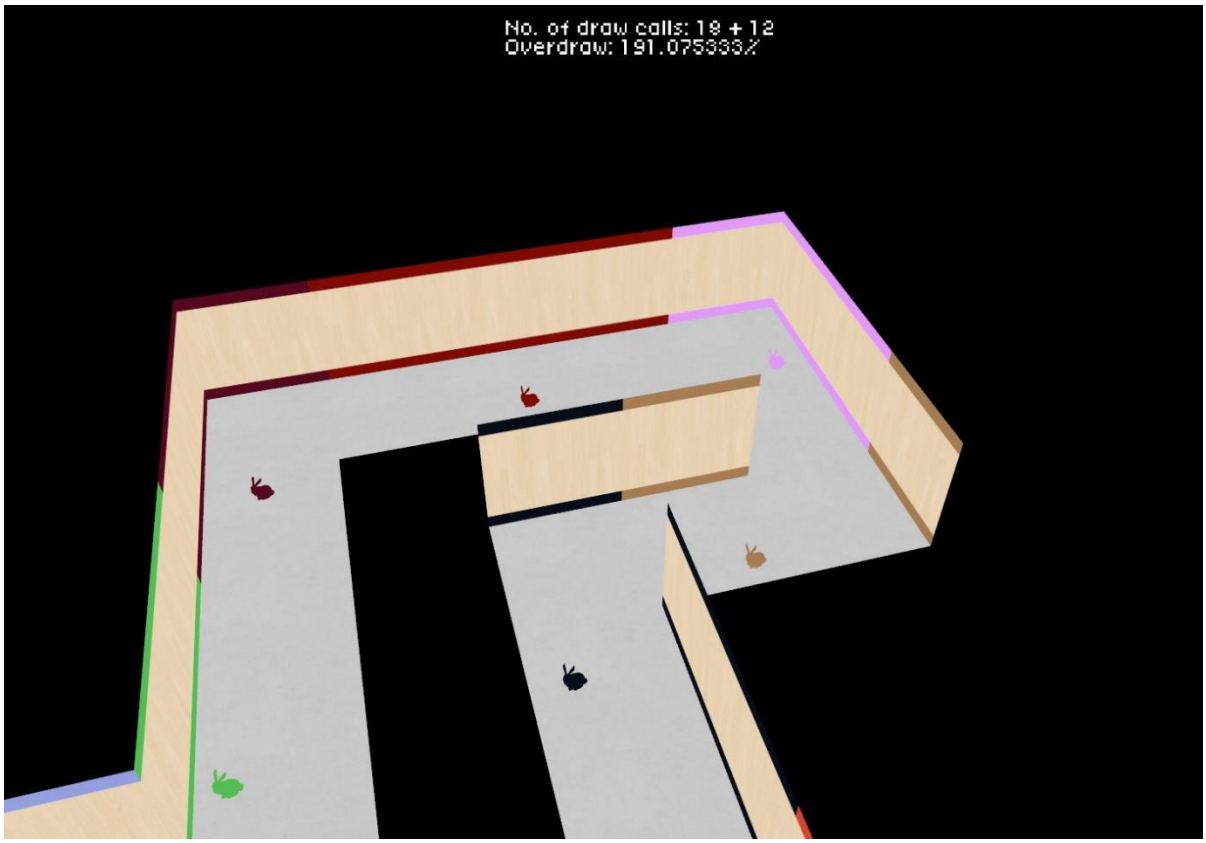
```
1  drawBSPTree(Node, position)
2     if isLeaf(Node)
3        Node.draw()
4        return

5     side = clasifyPoint(Node.divider, position)

6     if side is INFRONT or COINCIDING
7        drawBSPTree(Node.RightChild, position)
8        drawBSPTree(Node.LeftChild, position)
9     else
10       drawBSPTree(Node.LeftChild, position)
11       drawBSPTree(Node.RightChild, position)
```

Kód segment 6 – kreslení BSP stromu

Odvozeno z algoritmu (kód segment 10) prostým přehozením pořadí vykreslování pravých a levých potomků. V původním formátu se strom kreslí odzadu dopředu (4) a implementuje tak malířův algoritmus. Teď je ale pro nás výhodnější využít struktury BSP stromu ke kreslení odpředu dozadu, kde se využije maximálním způsobem výhod depth bufferu a ušetří se jeho neustálé přepisování.

1.5.2 BSP s occlusion query

Zde se využívá už vygenerovaných portálů a znalostí o sousedech. V každém snímku se začne vyhledáním listu, ve kterém se nacházíme, pomocí BSP stromu. Sousední listy pak iterativně kreslíme jen tehdy, když jsou viditelné. To zjistíme pomocí jednoduché occlusion query. Nevýhodou je nutnost čekání na její výsledek než se můžeme rozhodnout, zda kreslit dále.



Obrázek 3 - kreslení pomocí BSP stromu s occlusion query

Zde je vidět, že zadní kus bludiště není zobrazen, protože mezi ním a pozicí kamery je jeden nebo více listů, které nemají z pohledu kamery viditelné portály.

```
1 drawBSPTree_withQuery()

2    Node = queue.front()

3    if not visited(Node)
4       Node.draw()
5       Node.visited = true

6       foreach unvisitedNeighbour in Node
7          issueQuery(portal.connectedNode)

8          if portal.connectedNode is visible
9             queue.push(portal.connectedNode)
```

Kód segment 7 – kreslení BSP stromu s occlusion query

Ve svém principu zde jde o iterativní algoritmus s frontou listů. Na vstupu je fronta s listem, ve kterém se nachází kamera (vyhledáno pomocí BSP stromu). Pak se v cyklu volá tato funkce, dokud se fronta nevyprázdní.

Pokud již nebyl list navštíven (možné v případě kruhových chodeb apod.) list vykreslíme. Dále nad seznamem ještě nenavštívených sousedních listů provedeme occlusion query pro tyto listy a pokud jsou viditelné, umístíme je na konec fronty.

1.5.3 PVS

Kreslení s PVS je velmi jednoduché. Stačí po řadě vykreslit všechny listy uvedené v seznamu PVS. Vykreslují se sice i uzly momentálně neviditelné, jako například umístěné za kamerou, ale oproti pouhému BSP se jedná o významné urychlení.

Obrázek 4 - kreslení pomocí PVS

Tady se vykresluje jen pět listů, z toho dva z této perspektivy nejsou viditelné. Jeden, ve kterém se nachází kamera a jeden ještě více za ním.

```
1 drawPVSet(position)

2    Node = getNode(RootNode, position)

3    foreach node in Node.PVS
4       node.draw()
```
Kód segment 8 – kreslení PVS

Zde je funkce pro vykreslení PVS. Díky statickému charakteru předpočítané potenciální viditelnosti je extrémně jednoduchá a skládá se ze zjištění listu, ve kterém se nachází kamera, a vykreslení jeho PVS.

1.5.4 PVS s occlusion query

Zde se jedná o vylepšení předchozího způsobu o bufferované occlusion query, kdy se v daném snímku vykreslí jen ty listy, jejichž portály byly minulý snímek viditelné.

Obrázek 5 - kreslení pomocí PVS s occlusion query

Tento obrázek je vizuálně shodný s předchozím, ale lze si všimnout, že se změnila čísla u horního okraje. Místo čísla 10 je zde 8 + 4. To značí, že list nacházející se za listem s kamerou se už nevykreslil. Jsou viditelné 3 listy plus list, ve kterém se nachází kamera – celkem 8 kreslících příkazů a 4 dotazy na viditelnost pro tři listy vepředu a jeden vzadu. Dotazy pro aktuální list se neprovádějí.

```
1 drawPVSet_withQuery(position)
2    Node = getNode(position)

3    Node.visible = true

4    foreach node in Node.PVS
5       if node is visible
6         node.draw()

7    foreach node in Node.PVS
8       issueQuery(node)
```
Kód segment 9 – kreslení PVS s occlusion query

Zde se stejně jako v předchozím případě nalezne list, ve kterém se nachází kamera. Uzel, ve kterém se nacházíme, můžeme vždy považovat za viditelný. Při prvním volání této funkce se následně vykreslí listy v PVS (předpokládáme, že defaultní stav příznaku visible listů je true). Tímto jsme zobrazili všechny occludery (tj. objekty potenciálně zastiňující jiné objekty). Teď při provedení occlusion query v každém listu v PVS získáme přesnou informaci, zda je list viditelný, či ne pro následující snímek.

U variant s occlusion query nastává problém při situacích, kdy se jediný portál v zorném poli nachází ve vzdálenosti menší než je vzdálenost blízké clipping plane, která samozřejmě zabrání vykreslení takového portálu a tím pádem je celý uzel označen jako neviditelný. Je proto nutné každý snímek otestovat zda se kamera nenachází v takovéto pozici a nevykreslení sousedního uzlu zabránit.

1.5.5 BSP kreslené odzadu

Tento způsob se liší od prvního (kód segment 6) procházením stromu v opačném pořadí a nepoužívá Z-buffer. Jedná se o využití malířova algoritmu a vizuální výsledek je shodný s prvním způsobem. Tento způsob je použit při měření overdaw, ale na naměřené hodnoty nemá z praktického hlediska vliv. Jedná se jen o demonstraci konceptu BSP stromu jako prostředku pro malířův algoritmus.

```
1 drawBSPTree(Node, position)
2    if isLeaf(Node)
3       Node.draw()
4       return

5    side = clasifyPoint(Node.divider, position)

6    if side is INFRONT or COINCIDING
7       drawBSPTree(Node.LeftChild, position)
8       drawBSPTree(Node.RightChild, position)
9    else
10      drawBSPTree(Node.RightChild, position)
11      drawBSPTree(Node.LeftChild, position)
```
Kód segment 10 – kreslení BSP stromu odzadu

Toto je algoritmus (4) použitý v původní práci zabývající se konceptem BSP stromů jako způsobu řešení problému viditelnosti.

1.5.6 PVS kreslené odzadu

Stejně jako předchozí způsob nepoužívá Z-buffer. Kreslení odzadu pomáhá lehce eliminovat vizuální chyby, ale kromě toho se nijak neliší od kreslení zepředu. PVS není vhodnou implementací pro malířův algoritmus a je zde jen pro účely měření overdraw. Také nemá vliv na naměřené hodnoty oproti PVS.

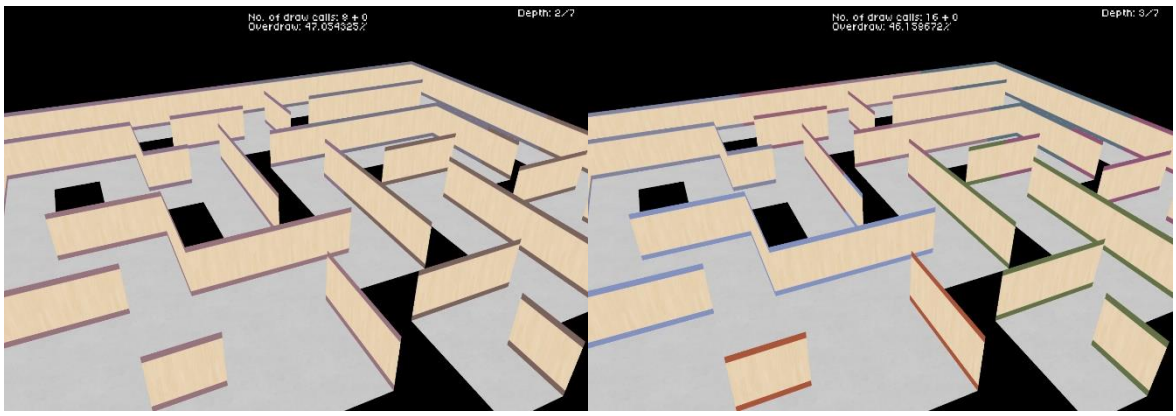
1.5.7 BSP kreslené po úrovních zanoření

Tady se vykresluje podobným způsobem jako čistý BSP strom, ale kromě ukazatele na uzel se v parametrech předává hloubka, kterou chceme dosáhnout, a při rekurzivním volání funkce se z ní odečítá. Tento způsob vykreslování se nepoužívá k získávání dat, ale pro demonstraci procesu dělení prostoru v BSP stromu a pro vizuální kontrolu.

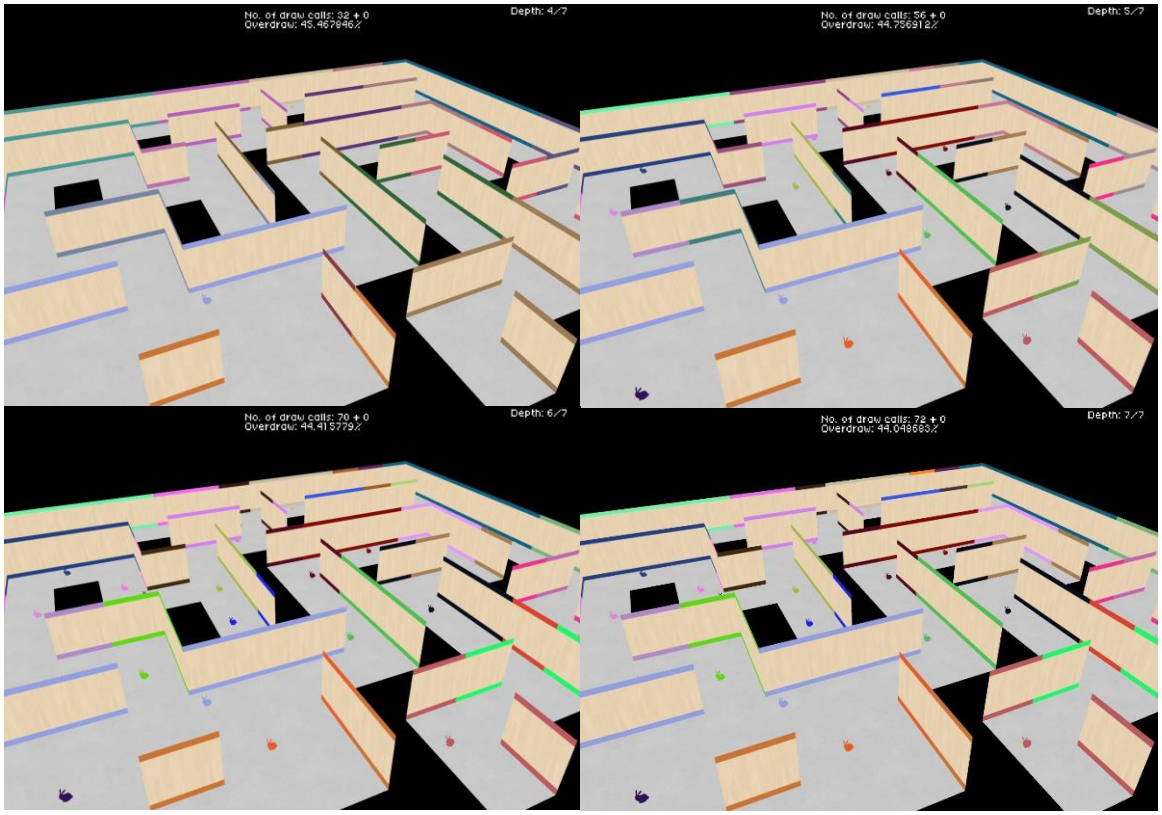
Každý uzel má svoji barvu, která se používá při kreslení. Pro účely tohoto způsobu se z barev listů udělá průměr a spolu s geometrií potomků se použije při kreslení jejich rodiče. Při provedení tohoto až k vrcholku stromu, získáme v kořenovém uzlu kompletní geometrii bludiště s velmi nevýraznou až šedivou barvou.

Při zvyšování zanoření se tato barva štěpí na své původní komponenty a jednotlivé uzly se rozjasňují, až se dosáhne nejvyšší granularity dělení a vykreslí se i se svým králíkem, který se směrem vzhůru nepropaguje.

Následující obrázky tento proces ilustrují na menším bludišti se sedmi úrovněmi. Série obrázků začíná úrovní zanoření 2. Při zanoření 2 je trochu patrné dělení horizontálně a pak vertikálně a v hloubce 3 už začínají být vidět jasnější barvy.



Dál v hloubce 4 je vidět první hotový list stromu a v hloubce 5 vypadá, že většina listů je již dokončených. Zde je také nejlépe díky barvám vidět samotné dělení odehrávající se například v pravém dolním rohu. Na obrázku s hloubkou 4 jsou vidět oblasti se světle hnědou a tmavě zelenou barvou, které ještě nejsou listy. V hloubce 5 je hezky vidět, jak se tmavě zelená rozdělila na světle zelenou vlevo a téměř černou vpravo, a také jak se světle hnědá rozdělila na červenou a zelenou. Zelená však stále ještě není konvexním uzlem, tvoří krátkou rohovou chodbu. V hloubce 6 je dosaženo definitivní podoby této části chodby se zářivou zelenou a jasnou červenou.



Obrázek 6 - sada obrázků - ilustrace dělení bludiště

2 Popis implementace

V implementaci především umisťování portálů bylo zavedeno několik změn pro zprůhlednění průběhu tohoto procesu a také kvůli problémům s přílišnou rekurzí u větších stromů.
Původní jedna velká rekurzivní funkce byla rozdělena do tří fází. V první fázi se vygenerují portály a umístí se do listů stromu. Ve druhé fázi se tyto portály ořežou geometrií listu a odstraní se části překrývající se s jednotlivými trojúhelníky a získáme jen polygony vyplňující mezery v geometrii listu. Nakonec se ještě zkontroluje, zda se nenachází více portálů přes sebe, případně zda více portálů nepokrývá jeden širší průchod a tyto se sloučí v jeden.

```
1 placePortals()
2    PortalQueue.push(RootNode->getPortal())
3    do {
4       portalPolygon = PortalQueue.back()
5       placePortal(RootNode, portalPolygon)
6    } while not PortalQueue.empty()
7    foreach leaf in NodeSet
8       leaf.processPortals()
9    mergeOverlappingPortals()
```

Kód segment 11 – řízení umisťování portálů

2.1 Umístění portálů

Jde o variaci na algoritmus v sekci 1.3.1 upravenou pro praktické využití s frontami místo problematické rekurze. Tato funkce má tři hlavní části.

V první se v uzlech generují portály a umísťují se na konec fronty s portály ke zpracování. K tomuto dochází jen v případě, že uzel není listem a pokud dělicí rovina ještě pro vygenerování portálu nebyla použita. Vytvoří se portálový polygon lehce přesahující bounding box aktuálního uzlu a vloží se na konec fronty s portály ve stromu (ta je odlišná od front s portály v listech). V případě, že příchozí polygon kříží dělicí rovinu aktuálního uzlu, tak se rozdělí podle této roviny a rozdělené části se vloží na konec stromové fronty s portály a funkce se vrátí.

```
1 placePortal(Node, portalPolygon)

2    if not node->isLeaf())

3       if not Node.divider.usedAsPortal

4          portal = Node.getPortal()

5          PortalQueue.push_back(portal)


6       side = calculateSide(Node.divider, portalPolygon)

7       if side is SPANNING

8          polygon front, back

9          splitPolygon(node.divider, portalPolygon, front, back)

10         PortalQueue.push_back(front)
11         PortalQueue.push_back(back)

12         return

13      if side is INFRONT
14         placePortal(node->RightChild, portalPolygon)

15       else if side is BEHIND
16         placePortal(node->LeftChild, portalPolygon);

17       else if side is COINCIDING

18         portal2 = portalPolygon

19         placePortal(Node.RightChild, portalPolygon)
20         placePortal(Node.LeftChild, portal2)


21    else

22       Node.PortalQueue.push(portalPolygon)
```

Kód segment 12 – umisťování portálů

Druhá část této větve se používá pro rekurzivní směrování příchozího polygonu stromem. Když uzel není listem a portál neprotíná dělicí rovinu, pošle se níže stromem v závislosti na své poloze vůči dělicí rovině uzlu. Pokud je vepředu, je poslán pravému potomku, pokud vzadu, levému. V případě že portálový polygon tvoří stejnou rovinu s dělicí rovinou, musí být poslán oběma potomkům.

Třetí částí je pouze uložení příchozího portálu v případě, že dosáhl listu stromu. Výsledkem je, že ze všech dělicích rovin uzlů byly vygenerovány portálové polygony, které se dělily všemi dělicími rovinami po cestě a skončily ve frontách listů.

V původní verzi tohoto algoritmu se nevyužívaly fronty a na místě operací s frontou se tato funkce znovu rekurzivně volala. To u většího množství generovaných portálů vedlo k zahlcení stacku. Teď se rekurze využívá jen k směrování portálu stromem a tento případ už nehrozí.

2.2 Výpočet PVS

Tato funkce operuje nad seznamem všech uzlů a listů a seznamem všech portálů uložených ve stromu a vytvořených během procesu generování BSP stromu a umisťování portálů. Tyto seznamy jsou pojmenovány `PortalSet` a `NodeSet`.

V první části tohoto algoritmu se najdou páry portálů, které sdílejí stejný prostor a tyto se uchovají v poli `immediateNeighbours a PVS` daného uzlu. Když známe bezprostřední sousedy všech listů, můžeme pokročit dál a pokusit se množinu potenciální viditelnosti co nejvíce rozšířit.

```
1 computePVS(){

2    foreach portal in PortalSet
3       foreach unconnectedPortal in PortalSet
4          if portal is unconnectedPortal
5             continue

6          if shareSpace(portal, unconnectedPortal)
7             connectPortals(portal, unconnectedPortal)


8    foreach leaf in NodeSet
9       foreach portal in leaf
10         leaf.immediateNeighbours.add(portal.connectedTo)

11      leaf.PVS.add(immediateNeighbours)


12   foreach leaf in NodeSet

13      list potentialNeighbours

14      foreach neighbour in leaf.immediateNeighbours

15         potentialNeighbour.add(neighbour.immediateNeighbours)


16      foreach potentialNeighbour in potentialNeighbours

17         visible = areNodesVisible(leaf, potentialNeighbour)

18         if visible
19            leaf.PVS.add(potentialNeighbour)
20            potentialNeighbours.add(potentialNeighbour.immediateNeighbours)
```

Kód segment 13 – výpočet PVS

Takže se po řadě pro všechny listy:

Vytvoří množina potenciálních sousedů, kterou ze začátku tvoří množiny bezprostředních sousedů bezprostředních sousedů aktuálního zpracovávaného listu. Ta se pak za běhu neduplicitně rozšiřuje.

Pro každého potenciálního souseda se otestuje viditelnost mezi aktuálním listem a tímto sousedem. V případě viditelnosti se tento přidá do PVS aktuálního listu a jeho bezprostřední sousedé do seznamu potenciálních sousedů (8).

2.3 Funkce areNodesVisible

Dva listy jsou viditelné za předpokladu, že paprsek vedený z portálu jednoho do portálu druhého neprotíná žádnou geometrii listů mezi těmito dvěma portály a neviditelné, když žádný paprsek vedený z jednoho portálu ke druhému neprojde bez protnutí nějaké geometrie (8). A protože jeden paprsek často nemusí stačit je potřeba vygenerovat více bodů pro každý testovaný portál (5) – funkce `getPoints`.

Pro každou dvojici těchto bodů se projde množina potenciálně viditelných uzlů zpracovávaného listu a testuje se, zda tento parsek – úsečka něco protne. Pokud se jedné podaří projít všemi potenciálně viditelnými listy bez kolize, můžeme s jistotou prohlásit, že testované listy jsou navzájem viditelné. V případě že všechny kombinace koncových bodů vyprodukovaly kolizi, vrátí se defaultní hodnota false. Aby se tento proces alespoň trochu urychlil, zejména v případě bludišť s velkým počtem potenciálně viditelných sousedů, provádí se před samotným testováním geometrie jednodušší zkouška. Každý list má spočítané své extrémy a z nich vytvoříme čtyři body tvořící čtverec v rovině X-Z. U těchto bodů jednoduše otestujeme, na které straně paprsku se nachází, a pokud je výsledek pro všechny stejný, můžeme testování geometrie tohoto listu přeskočit, protože bounding box tohoto listu neprotíná vytvořený paprsek.

```
1 areNodesVisible(current_leaf, potentialNeighbour)

2    foreach portal1 in current_leaf
3       portal1_points = portal1.getPoints()

4    foreach portal2 in potentialNeighbour
5       portal2_points = getPoints(portal2_points)


6       foreach point1 in portal1_points
7       foreach point2 in portal2_points

8          hit = false;

9          foreach Node in current_leaf.PVNodes

10            if Node is potentialNeighbour
11               continue

12            start(point1.XZ)
13            end(point2.XZ)

14            pointA = sideOfLine(start, end, vec2(Node.xmin, Node.zmax))
15            pointB = sideOfLine(start, end, vec2(Node.xmax, Node.zmax))

16            pointC = sideOfLine(start, end, vec2(Node.xmax, Node.zmin))
17            pointD = sideOfLine(start, end, vec2(Node.xmin, Node.zmin))

18            if all are on the same side
19               continue


20            hit = hit or PV_Node->rayIntersectsSomething(point1, point2)

21         if not hit      //if at least once something didn't hit, it's visible
22            return true

23 return false
```

Kód segment 14 – výpočet viditelnosti

2.4 Průchod kamerou a funkce loadNextPosition

Na začátku je nutné ručně procházet bludištěm a vytvořit seznam pozic kamery, targetů (v podstatě směrů pohledu) a časů od začátku pohybu. Tyto se uloží do souboru se jménem bludiště a při novém spuštění programu se stejným jménem bludiště je tento soubor načten třídou kamery. Pak lze na základě času volat funkci kamery `loadNextPosition(float valueToInterpolate)`, s časovou hodnotou, která se pak použije pro výpočet konkrétní pozice a targetu pomocí Catmull-Rom splinu.

Pro účely měření, je ale takový způsob ne zcela vhodný. Proto se po načtení souboru s průchodem kamery zavolá ještě `camera.convertToFrames(10000.0f),` která načtené časové hodnoty přepočítá na odpovídající čísla snímků, s tím, že poslední časová hodnota odpovídá číslu snímku 10 000. Následně se funkce `loadNextPosition` volá každý snímek s aktuálním číslem snímku a tímto je zajištěno, že každý snímek má konkrétní pozici v bludišti nezávisle na efektivitě použitého způsobu kreslení. Trasa pohybu je uložena ve struktuře `movement` uvnitř třídy `Camera`. Pole `times` může obsahovat čísla snímků nebo časové hodnoty. Na funkci `loadNextPosition` to nemá vliv.

```
1 struct {
2       CRSpline positions
3       CRSpline targets
4       list times
5    } movement
```

Kód segment 14 – struktura s cestou kamery

```
1 loadNextPosition(valueToInterpolate)

2    current_index = getCurrentIndex()

3    if current_index + 1 != movement.times.size()

4       segment_duration =
                    movement.times[current_index + 1] - movement.times[current_index]


5       percentage_of_current_segment_travelled =
              (valueToInterpolate - movement.times[current_index]) / segment_duration


6       delta = movement.positions.delta()   //is number between 0.0 and 1.0

7       normalized_distance =
             delta * current_index + delta * percentage_of_current_segement_travelled


8       position = movement.positions.getInterpolatedSplinePoint(normalized_distance)
9       target = movement.targets.getInterpolatedSplinePoint(normalized_distance)
10    else
11      glutLeaveMainLoop()
```

Kód segment 15 – výpočet následující pozice kamery

Kamera si na začátku zjistí, kde v poli s dráhou se nachází; přesněji které pozice v poli s dráhou naposledy dosáhla. Protože se pracuje s naposledy dosaženým bodem a bodem následujícím je nutné kontrolovat přesažení mezí. Jakmile jsme zorientovaní, můžeme pokračovat ve výpočtu. Délka segmentu (`segment_duration`) je prostý rozdíl mezi časovou (snímkovou) hodnotou posledního dosaženého bodu a následujícího. Procento procestované v aktuálním segmentu

(`percentage_of_current_segment_travelled`) je poměr mezi vstupní hodnotou mínus poslední dosažené hodnoty ku vypočítané délce segmentu.

Delta je v podstatě konstanta rovnající se 1 podělené počtem pozic nebo časů nebo targetů (tento počet je shodný). Teď máme vše potřebné k poslednímu kroku. Pro interpolaci pozic splinem potřebujeme parametr t v rozsahu 0 – 1. Tuto normalizovanou vzdálenost teď získáme vynásobením `delta * current_index.` Tímto máme normalizovanou vzdálenost posledního dosaženého bodu. Dále přičteme `delta * percentage_of_current_segement_travelled` a máme normalizovanou vzdálenost naší vstupní hodnoty pro použití k interpolaci pozice Catmull-Rom splinem.
Pokud se dojde na konec, ukončí se zobrazovací smyčka, aby se zachytily výsledky jen těchto 10 000 snímků.

2.1 Použité knihovny a prostředky

GLSDK (Unofficial OpenGL Software Development Kit) je sbírkou multiplatformních knihoven a nástrojů pro usnadnění práce s OpenGL napsaných převážně v C++. Každá komponenta má svou vlastní licenci, která je přibližná MIT licenci.
Využívané komponenty jsou GL Load pro načtení funkcí OpenGL, FreeGLUT pro snadné implementování zobrazovací smyčky a použití periferií a OpenGL Mathematics pro definice typů a operací přímo kompatibilních s OpenGL.[1]

K práci na projektu bylo použito prostředí VS2012 s jazykem C++11. K dispozici na datovém disku je projektový soubor, který potřebuje ke svému spuštění tuto verzi nebo vyšší.

Pro simulaci trochu reálnější zátěže (pouhá geometrie bludiště je extrémně jednoduchá), se v každém zobrazovaném listu kreslí i zdecimovaný model Stanford bunny (9) s asi 5000 trojúhelníky načítaném ze souboru formátu obj[2].

Dále pro zobrazování informací na obrazovku jako je help, aktuální hodnota FPS, počet vykreslených snímků atp. byla vytvořena knihovna načítající TrueType font. Lze využít jak pro statický text (jako help), tak pro dynamické informace (jako FPS atp.), bez většího vlivu na celkový výkon zejména pomocí obměny měnících se dat jen po uplynutí určité doby či diskrétní změně dat.

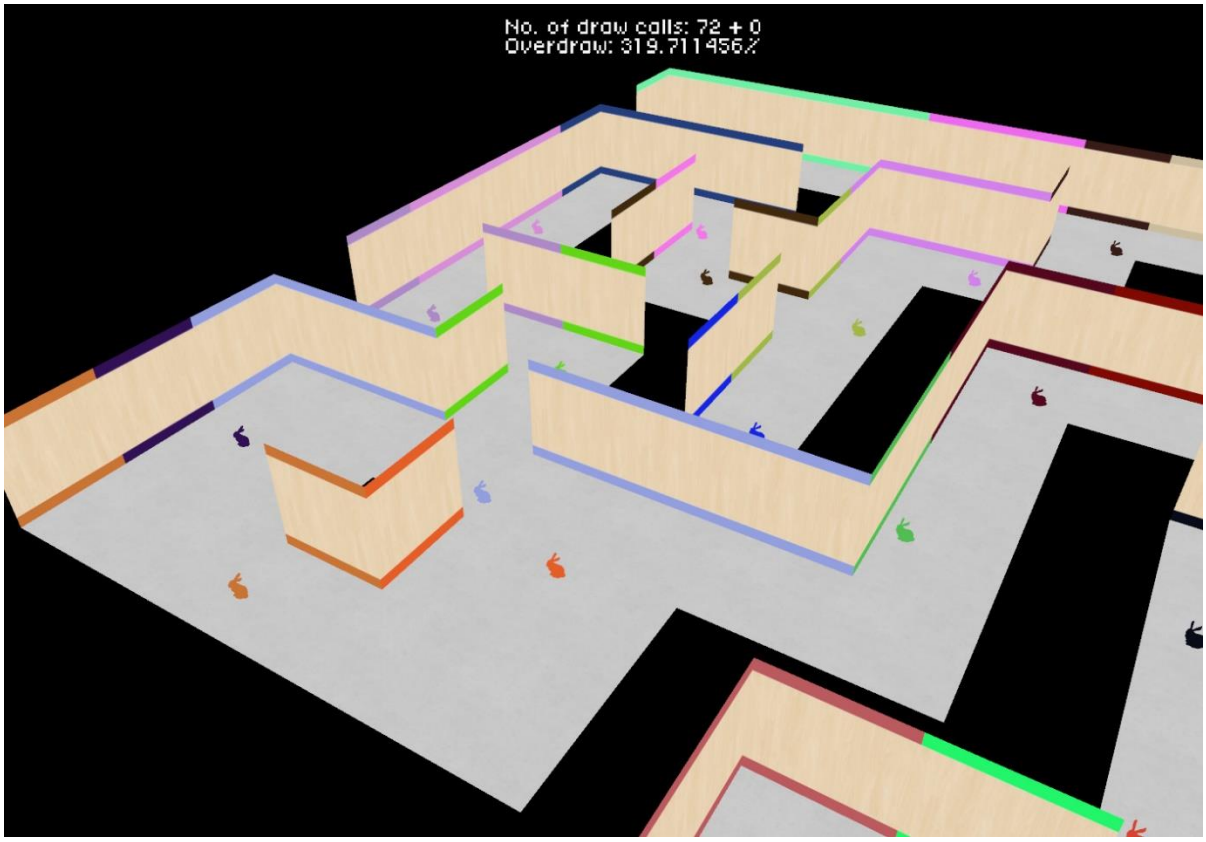
[1] http://glsdk.sourceforge.net/docs/html/index.html
[2] http://graphics.stanford.edu/~mdfisher/Data/Meshes/bunny.obj

3 Testy a měření

Všechny měření byly provedeny s konstatním počtem vykreslených snímků. Na vstupu je seznam bodů průchodu kamery bludištěm. Následně se zachycené časy přepočítají na čísla snímků s danou horní hodnotou (zde konkrétně 10000, respektive 10002 – dané implementací interpolace). Každý snímek se interpoluje pozice kamery na základě čísla aktuálního snímku. Tím se dosáhne toho, že vykreslovaná množina scén je vždy stejná bez ohledu na metodě vykreslování nebo časové náročnosti. Měření byla provedena na počítači vybaveném procesorem Intel Core2Duo E8400@3200Mhz a grafickou kartou NVidia GeForce 275.

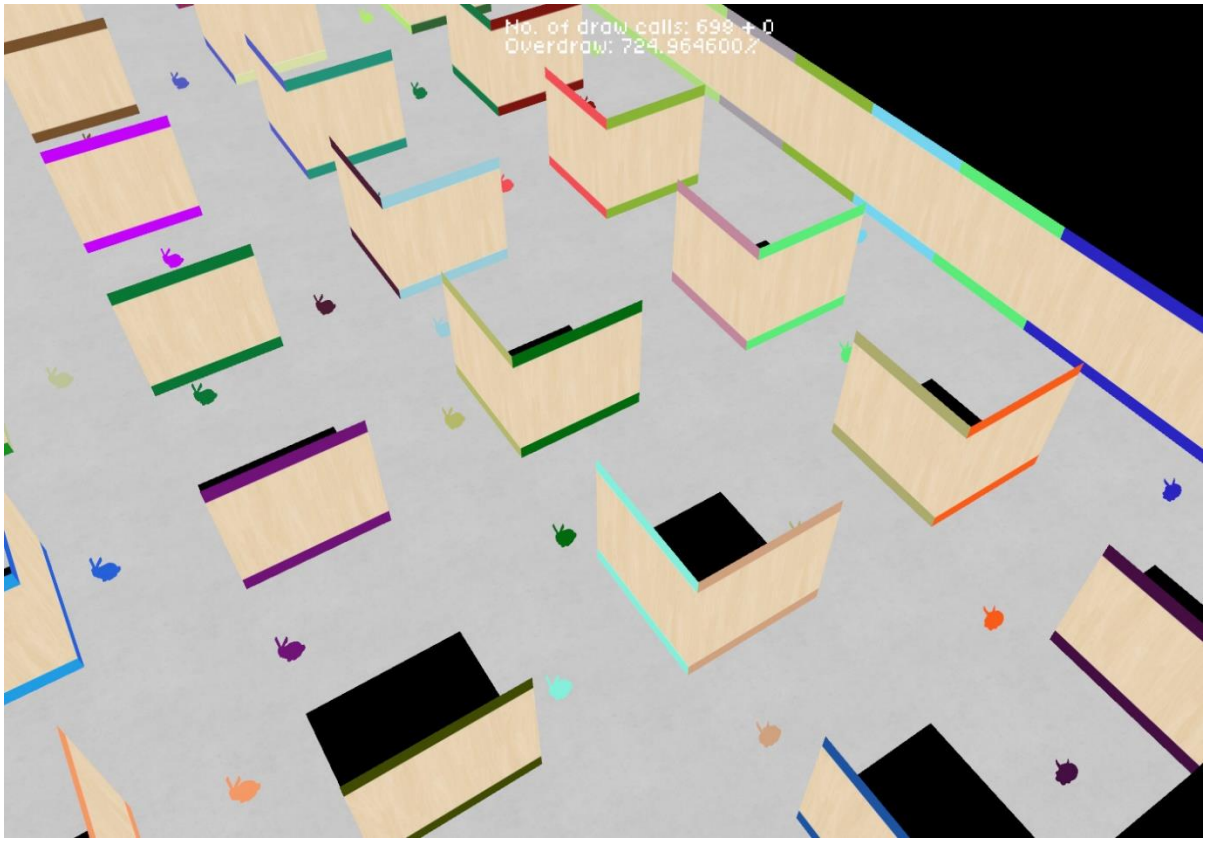
3.1 Typy použitých bludišť a sbíraná data

Typ regular je bludiště charakteristické chodbami s poměrně nízkým počtem rozcestí a občasným větvením. Toto je nejlepší případ vzhledem k náročnosti kreslení jinými způsoby než BSP stromu.



Obrázek 7 - ilustrace typu bludiště – regular

Typ columns značí bludiště tvořené jednou velkou místností se sloupy tvořícími pravidelnou mříž. Toto je nejhorší možný případ co se týče velikosti PVS a nutného množství prováděných visibility query.



Obrázek 8 - ilustrace typu bludiště - columns

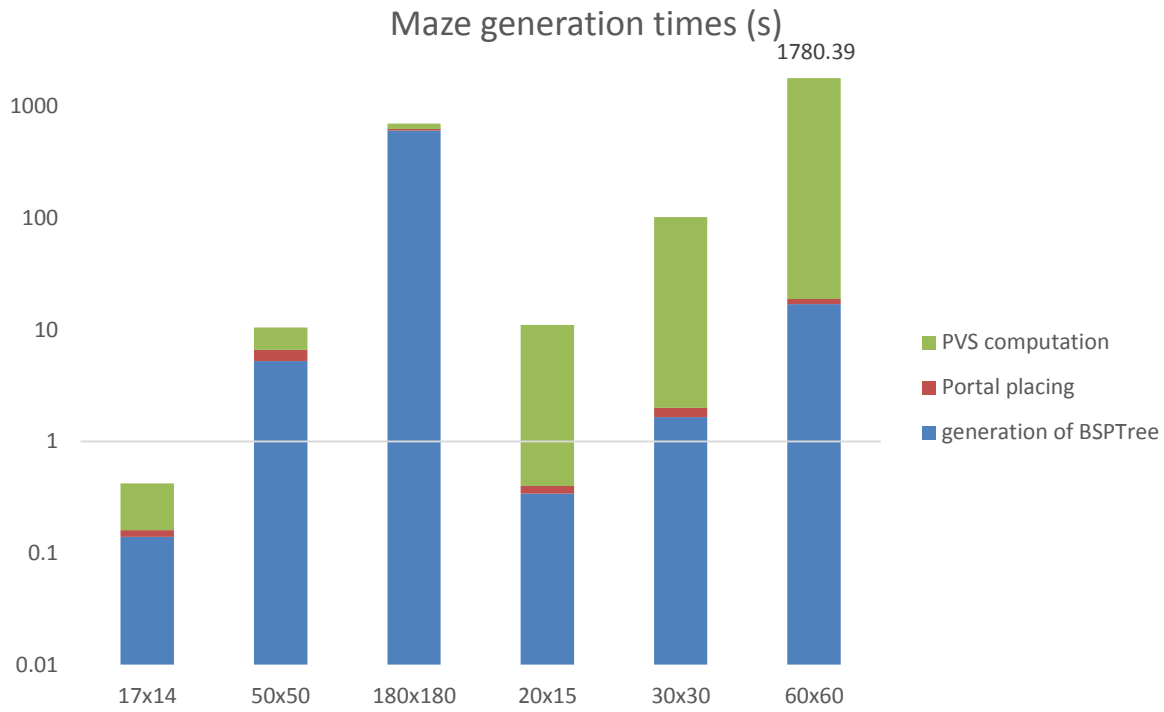
Při ukončování programu se vypíšou následující statistiky, které jsou zdrojem veškerých dat použitých v následujících grafech.

seconds elapsed	Množství vteřin od začátku běhu hlavní smyčky programu.
seconds spent drawing tree	Kolik času se strávilo ve funkci draw během běhu programu.
% of overall display time	Přepočteno na procenta
BSP avg draw calls:	Průměrný počet volání funkce typu glDraw*
BSP min draw calls:	Nejmenší počet volání funkce typu glDraw*
BSP max draw calls:	Nejvyšší počet volání funkce typu glDraw*
Overall samples passed:	Počet vzorků, které prošly depth testem.
Overdraw:	Poměrná hodnota mezi počtem vzorků ku počtu snímků a rozlišení okna, do kterého se vykresluje.
Avg FPS:	Průměrný počet snímků za vteřinu.
Avg render time:	Průměrný čas na snímek.
Frames rendered:	Počet snímků při běhu programu.

3.1.1 Parametry použitých bludišť.

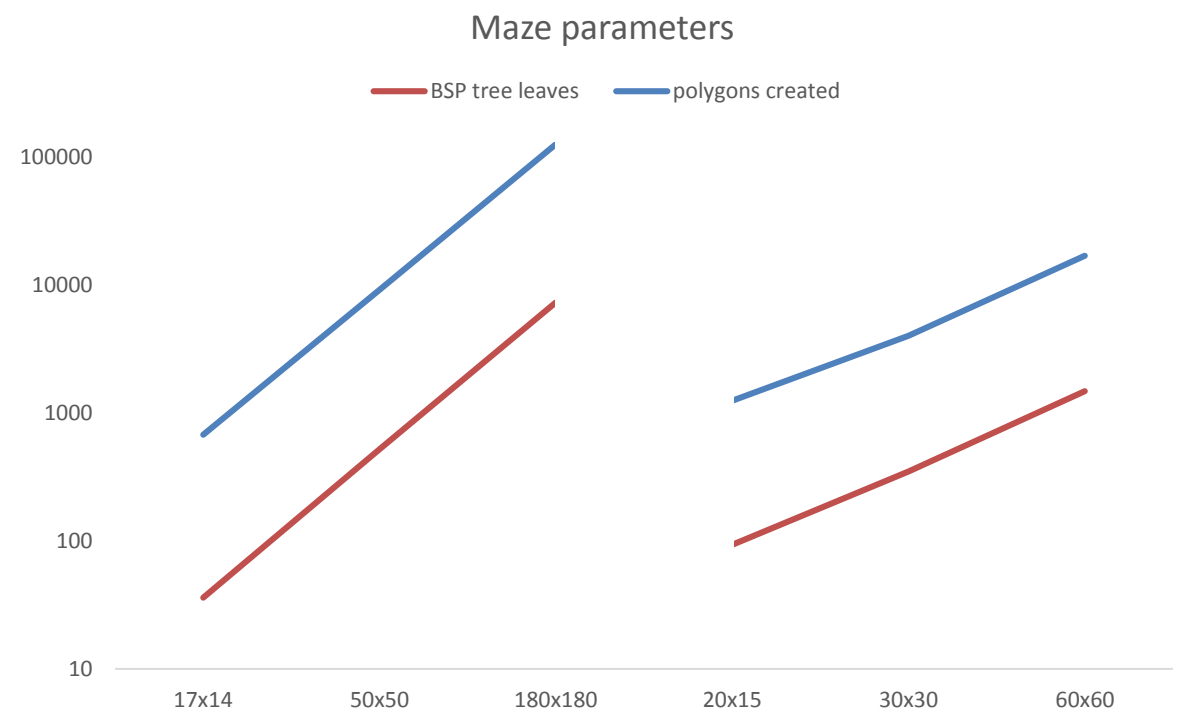
Pro účely testování byla použita šestice bludišť. Tři typu regular - 17x14, 50x50, 180x180 - a tři typu columns – 20x15, 30x30 a 60x60. V tomto pořadí jsou i uvedeny ve všech grafech.

3.2 Generování bludiště



Obrázek 9 - graf s časy generování bludišť

Zde je vidět kumulativní náročnost jednotlivých fází generování BSP stromu a počítání PVS s logaritmickým měřítkem. Zatímco u obyčejného bludiště je dominantním limitujícím faktorem množství geometrie při generování stromu BSP (konkrétně funkce nalezení dělicí roviny), u bludiště se sloupy se velmi rychle projeví náročnost výpočtů viditelnosti. Oproti těmito dvěma hlavním faktorům je generování a umisťování portálů časově téměř zanedbatelné.

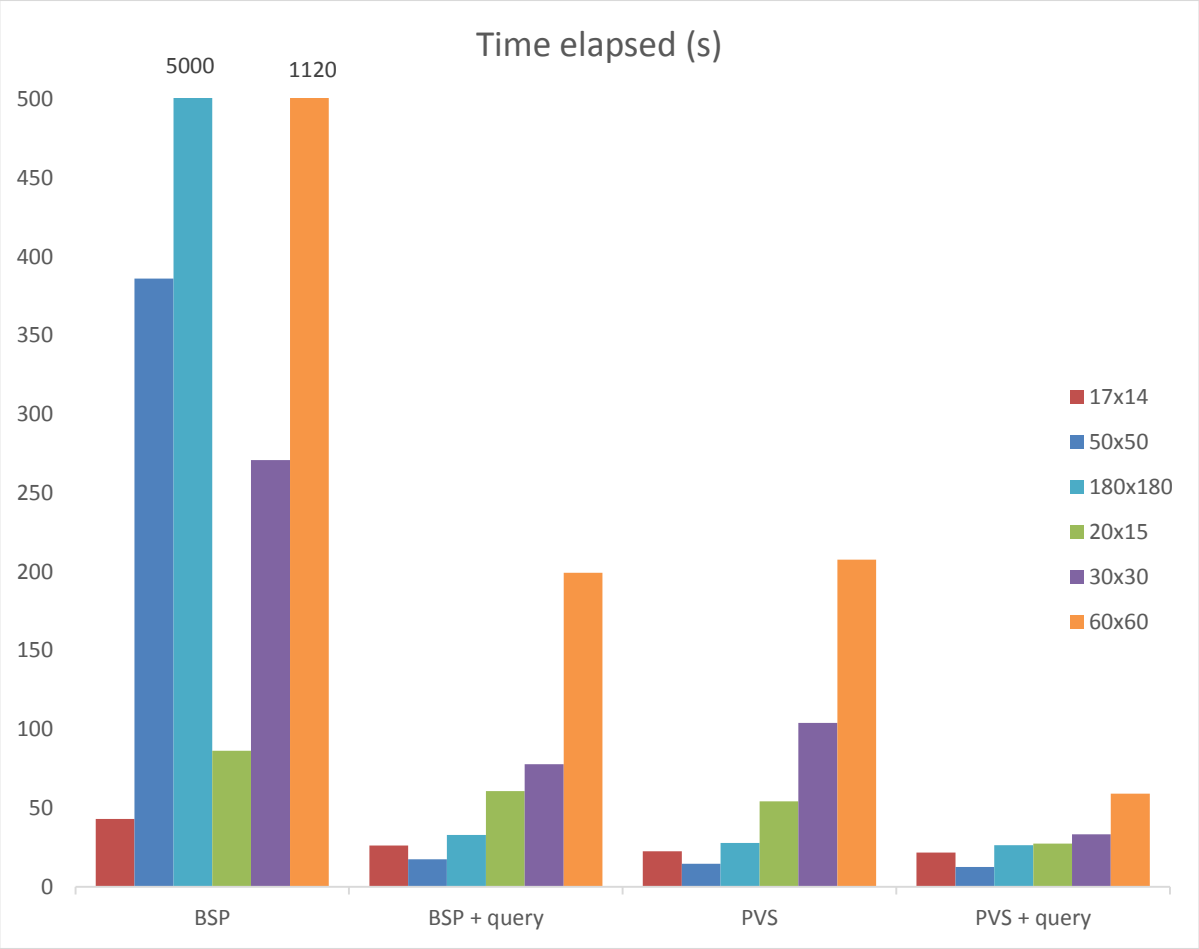


Obrázek 10 - graf s parametry bludišť

Zde jsou zobrazeny (také na logaritmické ose) počty listů a trojúhelníků v jednotlivých bludištích. Počty trojúhelníků zahrnují jen geometrii samotného bludiště; vykreslovaný Stanford bunny v každém listu se do tohoto čísla nezapočítává.

V kontextu předchozího grafu můžeme pozorovat, že sklon čar ilustrujících velikost bludiště zhruba odpovídá nárůstu časové složky generování BSP stromu, a je také vidět, že výpočet PVS je mnohem více ovlivněn uspořádáním geometrie než samou velikostí bludiště.

3.3 Uplynulý čas



Obrázek 11 - graf s časy průchodů

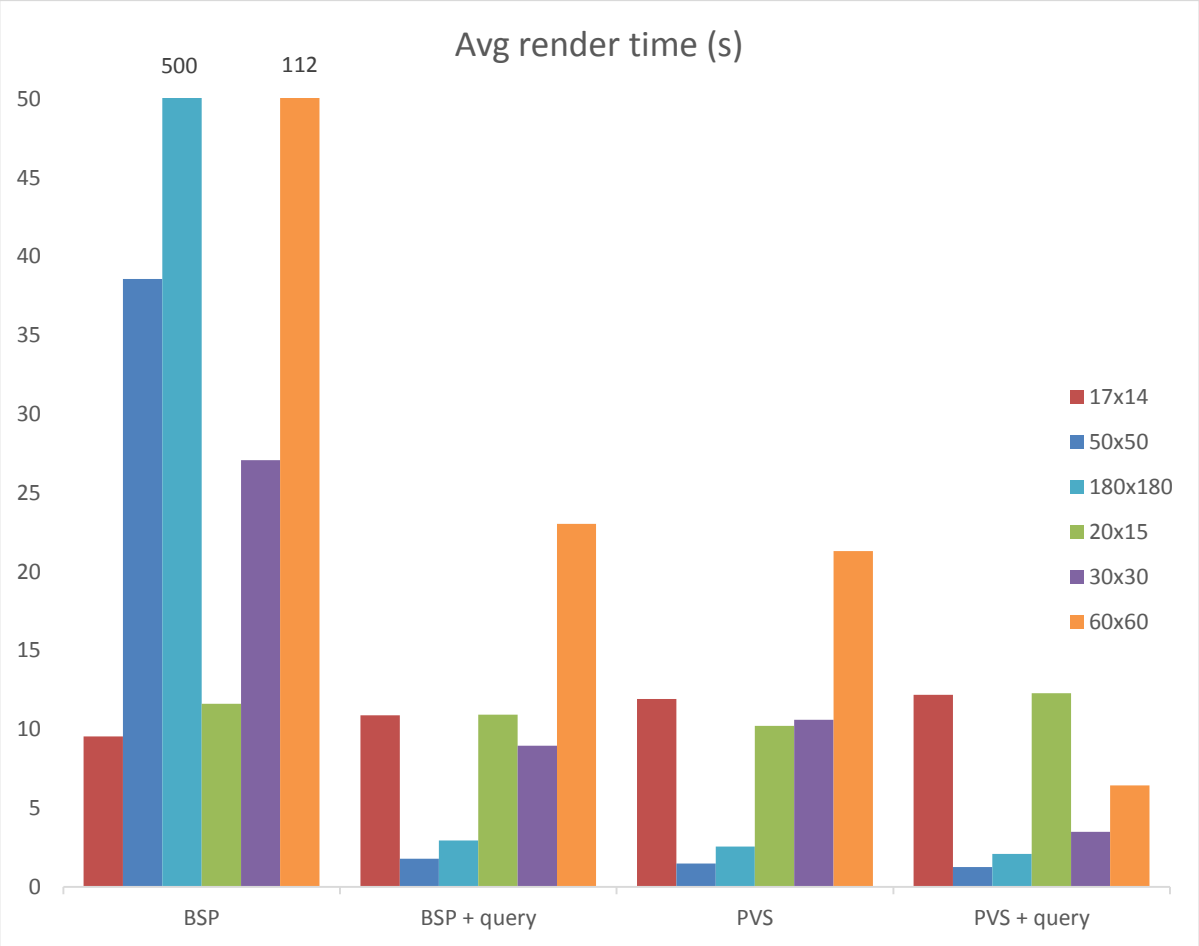
Uběhnutý čas je asi nejhrubější měřenou hodnotou. Zahrnuje v sobě vše od výpočtů pozice kamery po samotné kreslení příslušnou metodou. Z grafu je patrné, že vykreslování všeho se velmi rychle stává neúnosně časově náročným. Hodnota u bludiště 180x180 a 60x60 je pouze odhadovaná na základě přibližného času snímku a známého konstantního počtu snímků.

Už zavedení occlusion query při kreslení BSP stromu přináší znatelné zrychlení ve všech typech bludišť, ale protože se musí na výsledek každé query čekat, při větším množství dotazů to vede k čím dál znatelnějším prostojům a zbytečnému zdržování, jak je nejvýrazněji patrné u bludiště 60x60.

Při přestupu na kreslení s PVS se eliminuje veškeré čekání na occlusion query, ale zase to znamená, že se vykresluje vše co je v tomto PVS bez ohledu, zda je to viditelné. Proto u obyčejných bludišť je zrychlení malé a u bludišť se sloupy a velkým množstvím potenciálně viditelných sousedů spíše naopak.

Nejlepších časů je dosaženo kombinací PVS a occlusion queries. Máme výhodu omezené množiny, kterou je nutné testovat, a může se použít bufferovné occlusion query, která eliminuje zbytečné čekání.

3.1 Průměrný čas kreslení

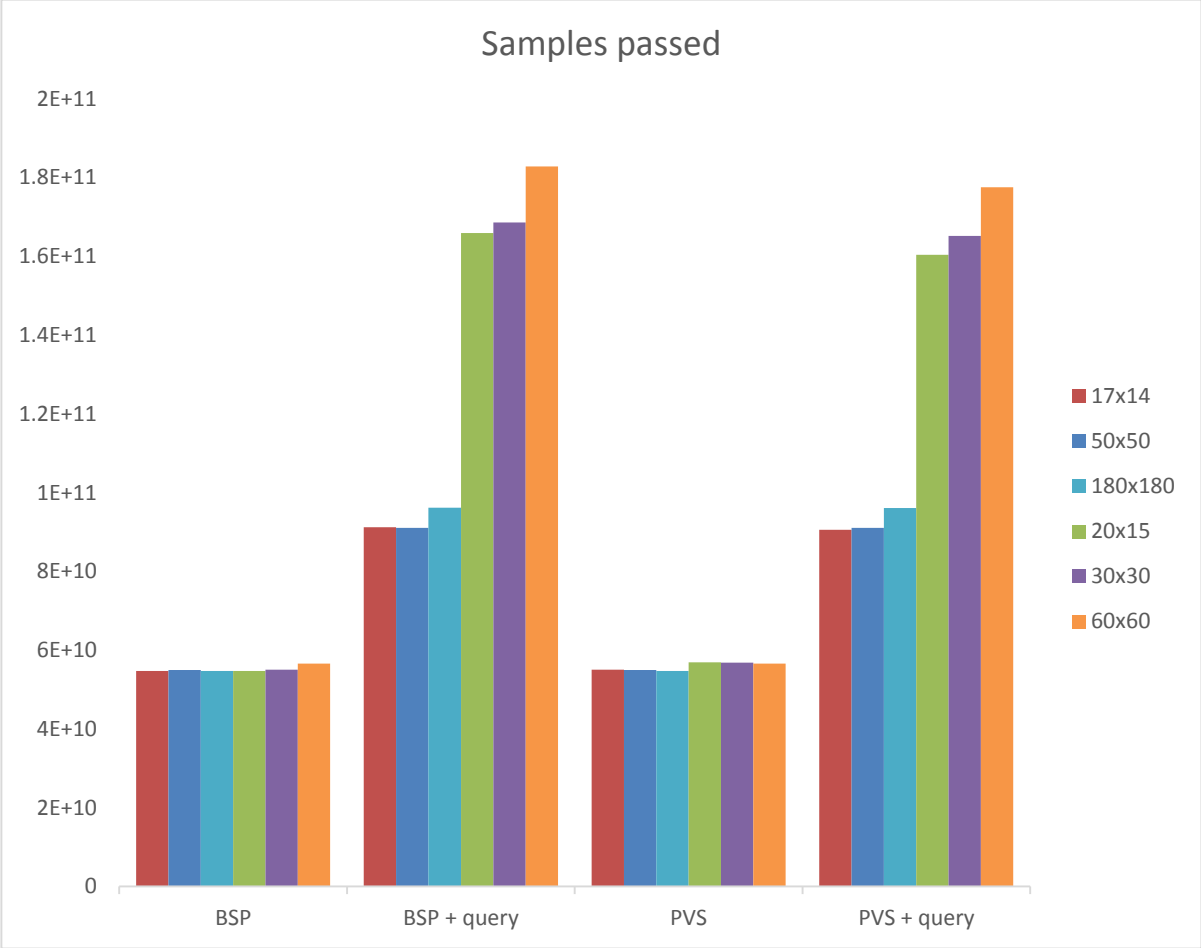


Obrázek 12 - graf s průměrnými časy na jeden snímek

Zde je vidět průměrný čas na jeden vykreslovací cyklus. Je také vidět, že u malých bludišť, konkrétně 17x14 a 20x15 dochází k nějakému zvláštnímu jevu, kde jejich naměřené časy neškálují podle očekávání a zůstávají víceméně konstantní. Je možné, že je to způsobeno nějakým mechanismem uvnitř gpu nebo ovladačů. Větší bludiště škálují podle očekávání. Zajímavé je, že tato skutečnost neovlivňuje hodnoty uvedené v sekci 3.3 – uplynulý čas. Nejspíše jde o jev způsobený ovladači.

Je také vhodné zdůraznit, že u způsobu PVS + query lze díky známé PVS na rozdíl od způsobu BSP + query, kde se na výsledek každé occlusion query musí čekat, aby se dosáhlo bezchybného obrazu, použít bufferované query bez čekání s prodlevou jen jeden snímek. Bez této optimalizace se rychlost kreslení od PVS a BSP + query příliš neliší a postrádá tak smysl.

3.2 Počty vzorků



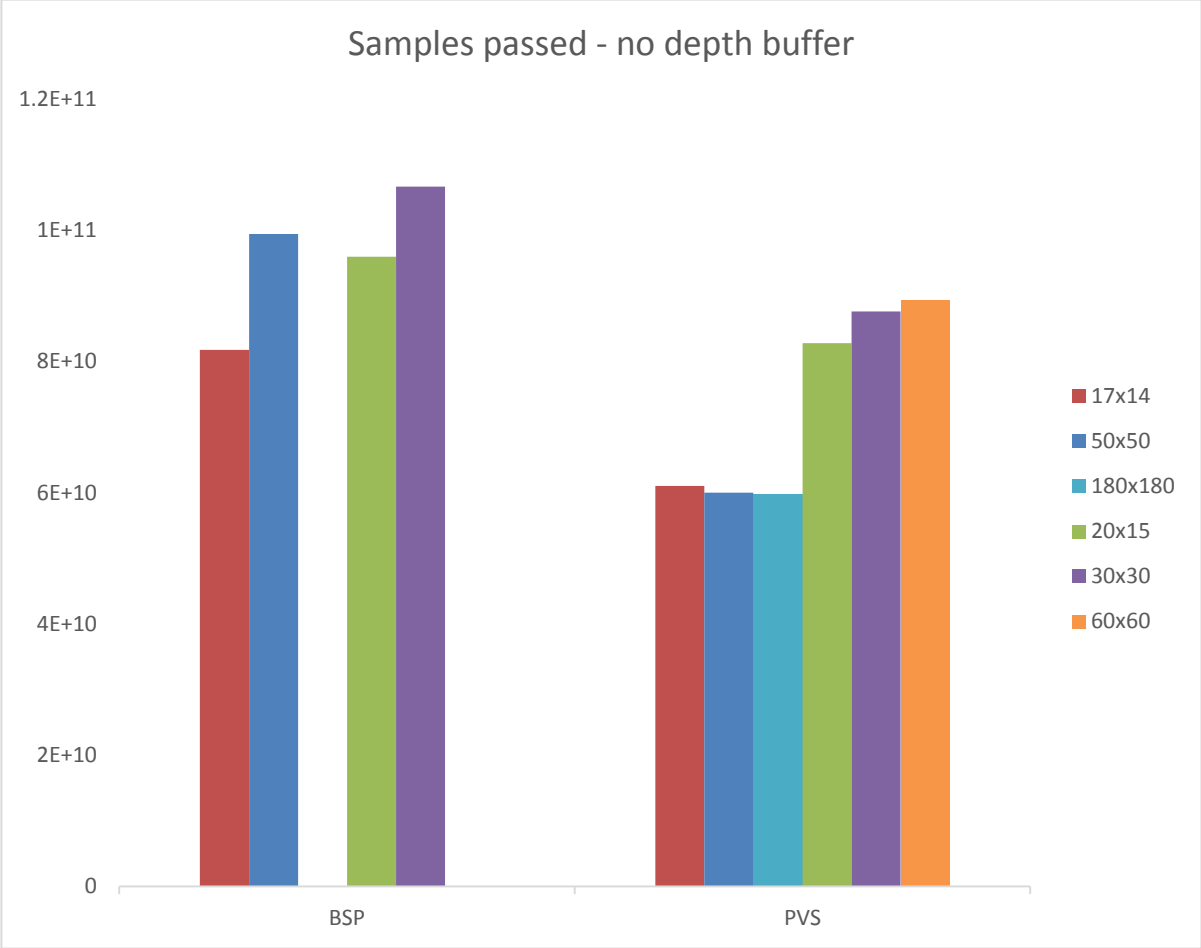
Obrázek 13 - graf s počty vzorků prošlých depth testem

Zde je vidět, že počet vzorků je víceméně konstantní, avšak jsou zde identifikovatelné 3 různé hladiny. Nejnižší hladina u BSP a PVS je daná především rozlišením okna a od toho se odvíjejícím počtu pixelů, které mohou být vzorkovány. Tuto hodnotu lze považovat za nejnižší možnou.
Druhá, střední, hladina je patrná u bludišť obyčejného typu, kde se už využívá occlusion query. Rozdíl mezi touto hladinou a tou základní tvoří vzorky vzešlé z vykreslování portálů pro occlusion query s vypnutým zapisováním do depth bufferu.

Třetí skupina je shodná s druhou co do původu vzorků, ale liší se typem bludiště, kde je např. u 30x30 možné dosáhnout až 200+ současných occlusion query. Protože je ale plocha portálu se vzdáleností stále menší, není přírůstek mezi 20x15 a 30x30 příliš znatelný.

Dále je ještě malý rozdíl mezi BSP s query a PVS s query. PVS s query je malinko efektivnější. To plyne ze způsobu kreslení a i z celkového principu PVS. Kde PVS ví, kde přestat testovat viditelnost portálů, BSP s query musí testovat vždy ještě jeden list navíc.

3.1 Počty vzorků bez depth testu



Obrázek 14 - graf s počty vzorků s vypnutým depth testem

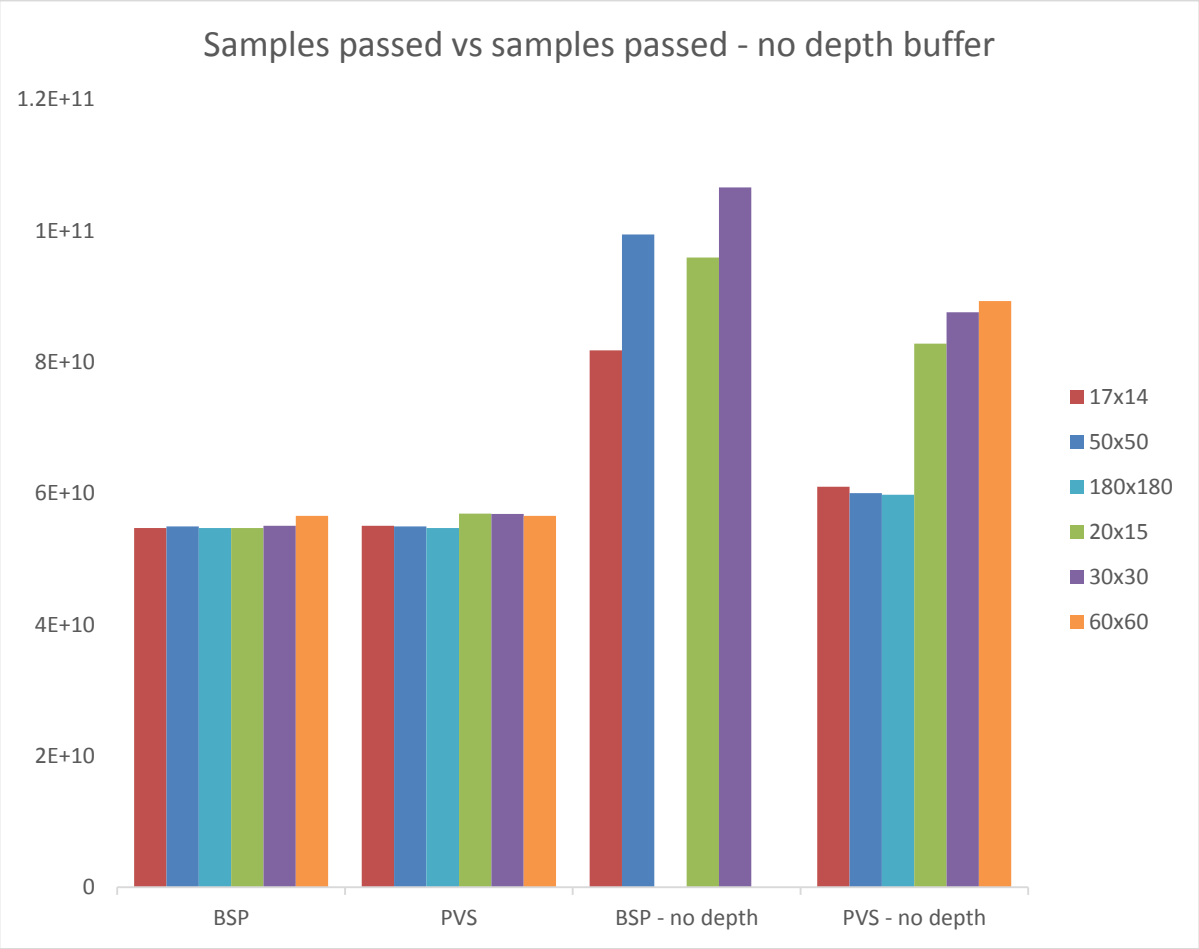
Zde je u BSP použita metoda kreslení odzadu a výsledný obraz vypadá správně. Jde zde ale jen o ověření funkce BSP stromu jako struktury umožňující použití malířova algoritmu. Na naměřené hodnoty tento rozdíl nemá vliv.

Znamená to také, že množství vzorků je závislé na velikosti bludiště, protože malířův algoritmus postupně kreslí odzadu a postupně překrývá vše s nižší hloubkou. Výsledným obrazem je, až co zůstane po vykreslení seřazených polygonů. U BSP data bludišť 180x180 a 60x60 chybí z důvodů nepraktické časové náročnosti (viz graf 3.3).

U metody BSP množství vzorků roste s velikostí bludiště ve své kategorii. To je dáno prostým množstvím geometrie, která se musí překreslit. U bludišť se sloupy jsou tyto hodnoty mírně vyšší oproti obyčejnému typu z důvodu efektivnějšího pokrytí prostoru bludiště geometrií, kde obyčejný typ má mnohem více prázdných míst.

U metody PVS je vidět hodnota u normálního typu bludišť, která je oproti BSP nižší, ale především neroste s velikostí bludiště. U bludišť se sloupy je úspora nižší z důvodu mnohem většího množství listů vedoucích jen za roh a tak navyšujících počet nutných vzorků.
Další malý pozvolný nárůst je dán zvětšující se maximální délkou viditelné chodby s viditelnými listy. Celkově se dá říct, že PVS značně uspoří množství vzorků, které je nutné posílat do depth buferu, i když v extrémních případech tato úspora není tak dramatická a pohybuje se lehce přes 10%.

3.2 Porovnání vlivu depth testu na počet vzorků



Obrázek 15 - graf porovnání vlivu depth testu na počet vzorků

BSP a BSP – no depth jasně ukazují že při vykreslování vší geometrie stoupá počet vzorků s počtem zobrazitelné geometrie i když ne příliš drasticky, pokud vezmeme v úvahu, že např. v bludišti 50x50 může být až 20 chodeb za sebou. Nárůst je v extrémních naměřených případech jen okolo 90%. Toto lze vysvětlit faktem, že s rostoucí vzdáleností je část překreslené části okna stále menší díky perspektivě a největší regiony překrývají právě jen nejbližší zobrazované listy stromu a navíc se odvrácené strany trojúhelníků nevykreslují, takže toto je také určitým faktorem.

Malý nárůst v PVS – no depth oproti PVS je u normálních bludišť v důsledku pár malých oblastí skrytých za rohem. Proto je nárůst jen v oblasti okolo 10%. U sloupovitých bludišť je nárůst dost znatelný. Je sice menší než u BSP – no depth, ale je k jeho hodnotám blíže než k hodnotám u normálních bludišť v PVS – no depth. Tady si lze představit, co se vlastně kreslí v PVS a v BSP. Zatímco BSP kreslí vše co je v zobrazitelném výseku prostoru definovaném kamerou, PVS kreslí jen určitou podmnožinu. A v případě bludišť se sloupy jsou tyto množiny často dost podobné (zejména pro nejbližší listy s největším významem na počet vzorků). Proto je v grafu vidět malé snížení počtu vzorků původem ze vzdálenějších zobrazovaných listů oproti BSP – no depth.

4 Závěr

Kreslení větších 3D bludišť než velmi malých vyžaduje nějakou formu urychlování. Jakou formu vybrat už do určité míry záleží na typu geometrie 3D světa.

4.1 Zhodnocení naměřených výsledků

Pro bludiště s malým počtem sousedů a vedlejších chodeb pohodlně dostačuje využití jednoduchých occlusion query a znalostí o sousedech v BSP stromu. Pro komplexnější svět s velkým množstvím occludujících objektů je toto řešení nepříliš dostatečné.

Využití předpočítaného PVS v každém listu přinese samo o sobě tu výhodu, že kromě malého urychlení nevyžaduje žádné další výpočty na grafické kartě ve formě occlusion query. U obyčejných bludišť je výpočet PVS poměrně rychlou záležitostí a proto se vždy vyplatí. U komplexních bludišť se výpočet PVS poněkud protáhne a samo o sobě se použití jen PVS u tohoto typu příliš nevyplatí. Je to ale nutné pro další krok, který přináší nejlepší výsledky ze všech použitých způsobů.

PVS s occlusion query poskytuje obyčejným bludištím ještě další malé zrychlení a komplexním bludištím zrychlení velmi znatelné. Je vždy zaručená minimální množina potenciálně viditelných listů, která se ještě dále zredukuje na listy opravdu viditelné. Vede to sice k asi trojnásobně vysokému využití depth bufferu, ale to je jeho zamýšlené využití při occlusion query.

Bylo dosaženo přepsáním algoritmů pro umisťování portálů na variantu s frontami větší flexibility, co se týče velikosti zpracovávaných bludišť, urychlení výpočtu PVS rychlou předběžnou eliminací listů ze zjišťování kolizí s paprskem.

Poměrně zajímavým, i když docela malým přínosem je také implementace query bufferu garantující nulové čekání na výsledky query s pouze jednosnímkovým zpožděním, což je při běhu aplikace zanedbatelné co se týče statistik a neviditelné co se týče vykreslovaného obrazu.

Byl položen objektivní, měřitelný a opakovatelný základ pro další vývoj a optimalizace. Každá další prováděná změna může být objektivně posouzena vzhledem k již implementovaným způsobům kreslení a shromážděným datům.

4.2 Další vývoj

Z pohledu dalšího vývoje by určitě bylo relativně jednoduché přidání detekce kolizí urychlené BSP stromem. BSP strom umožňuje velmi rychle dohledat příslušný list, ve kterém se objekt nachází a protože list obsahuje jen omezené množství geometrie, je detekce případné kolize výpočetně mnohem méně náročná.

Dalším krokem by mohlo být přidání osvětlování a stínování. Pomocí BSP stromu lze najít pozici osvětlovaného objektu ve stromu a použít světla pro stínování jen z daného listu nebo také ještě z jeho bezprostředních nebo ještě vzdálenějších sousedů, případně z celého PVS.

Pak by šlo využít nabytých poznatků k vytvoření velmi rozlehlého prostředí s trochu realističtější prostorovou dispozicí, jako jsou místnosti, chodby, křižovatky a dále to rozvinout v nějaké interaktivní prozkoumávání s minimapou.

Vymyšlení způsobu jak urychlit generování BSP stromu, konkrétně algoritmu na hledání vhodné dělící roviny, který s rostoucím množstvím geometrie velmi špatně škáluje. Je nutné i při nalezení vhodné dělicí roviny prvním pokusem ji sekvenčně porovnat s kompletní geometrií bludiště. Nalezení prvních pár dělicích rovin ve velmi velkých bludištích trvá většinu času běhu algoritmu, kde každé další dělení v podstromu trvá asi jen polovinu předchozího času.

Nebo obejití předchozího problému generováním bludiště po částech do více stromů, které by se pak spojily do jednoho, ale pak nastává problém přechodů mezi jednotlivými segmenty, který by bylo nutné řešit.

Lze také zvážit vytvoření PVS pro každý portál zvlášť a v kombinaci s occlusion query na jednotlivé portály listu si pak ušetřit dotazování na celé neviditelné větve.

5 Reference

1. **Segal, Mark a Akeley, Kurt.** OpenGL 4.4 specification. *OpenGL.org.* [Online] 19. 3 2014. http://www.opengl.org/registry/doc/glspec44.core.pdf.

2. **Sellers, Graham a Boudier, Pierre.** *OpenGL.org.* [Online] 25. 10 2010. http://www.opengl.org/registry/specs/AMD/conservative_depth.txt.

3. *Hierarchical Z-Buffer Visibility.* **Greene, Ned, Kass, Michael a Miller, Gavin.** 1993. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. stránky 231-238.

4. *On Visible Surface Generation by A Priori Tree Structures.* **Fuchs, Henry, Kedem, Zvi. M and Naylor, Bruce F.** New York : ACM, 1980. Proceedings of the 7th annual conference on Computer Graphics and interactive techniques. pp. 124-133.

5. *Binary Space Partioning Trees and Polygon.* **Ranta-Eskola, Samuel.** 2001.

6. *Visibility Computations in Densely Occluded Polyhedral Environments.* **Teller, Seth.** 1992.

7. *Distributed visibility culling technique for complex scene rendering.* **Lu, Tainchi a Chan, Chenghe.** 2005, Journal of Visual Languages and Computing archive, stránky 455-479.

8. *Visibility Preprocessing.* **Teller, Seth a Séquin, Carlo.** 1991, Proceedings of the 18th annual conference on Computer graphics and interactive techniques, stránky 61-70.

9. **Turk, Greg a Levoy, Marc.** [Online] 1994. http://www-graphics.stanford.edu/data/3Dscanrep/.

Seznam příloh na DVD

Příloha 1. Manuál
Příloha 2. Zdrojové texty
Příloha 3. Tabulka se zdrojovými daty